FROG & FRIENDS

PROJEKT-DOKUMENTATION

Mitwirkende:

Fabian Schwarz
- 76508 -

Lukas Vogrinc
- 77871 -

22. Januar 2022

Inhaltsverzeichnis

Abbildungsverzeichnis

Dieses Dokument ist Ninja Frog gewidmet, welcher im Laufe der Entwicklung dieses Spiels tausende Male auf brutalste Weise ums Leben gekommen ist...

Kapitel 1

Projektübersicht

1.1 Konzept

1.1.1 Projektidee

Bei der Ideenfindung zu unserem Projekt haben wir uns schnell darauf geeinigt einen 2D-Platformer in Unity zu erstellen, welcher sowohl alleine gespielt werden kann, als auch in einem kompetitiven lokalen Mehrspielermodus im Split-Screen mit drei weiteren Spielern. Steuerbar sollten die Charaktere der Spieler sowohl mit Tastatur als auch mit Gamepad sein. Die Idee für das Ziel des Spiels war es anfangs noch ein Level in möglichst kurzer Zeit abzuschließen. Im Mehrspielermodus wären also alle Spieler am selben Punkt gespawnt und es hätte der Spieler gewonnen, der als erstes das Ziel erreicht. Diese anfängliche Idee wurde von uns allerdings verworfen, da sich die Spieler (besonders im 3- oder 4-Spielermodus) nur gegenseitig behindern, wenn sie zur selben Zeit an der selben Stelle im Level das gleiche versuchen. Unsere neue Idee sah vor, einsammelbare Items mit unterschiedlichen Werten in der Spielwelt zu verteilen und das Ziel des Spiels darauf zu ändern, möglichst viele dieser Items in einer festlegbaren Rundenzeit zu sammeln. Der Spieler, der am Ende also die meisten Punkte eingesammelt hat, gewinnt das Spiel. Um den Spielern dies möglichst zu erschweren sollen sich in der Spielwelt diverse tödliche Gegner und Hindernisse befinden, die den Spieler im Falle seines Todes an den naheliegendsten Spawnpunkt versetzen, von welchem er fortfahren kann.

Um das Spiel von anderen 2D-Jump-n-Run-Spielen abzuheben, entschieden wir uns dafür neben der fröhlichen, heiteren Aufmachung des Spiels mit einer für das Spielgenre unüblichen Brutalität einen kompletten Gegensatz einzubauen. Zusätzlich kam uns im Verlauf des Projekts noch die Idee einen Spielkommentator einzubauen, welcher den Spieler nicht leiden kann und jede seiner Taten mit einem spöttischen Spruch kommentiert. So soll das Spiel mit seiner grafischen und musikalischen Aufmachung zunächst an ein kindgerechtes, buntes Platformer-Spiel erinnern und sich dem Spieler das wahre Gesicht des Spiels erst während des eigentlichen Spielens offenbaren.

1.1.2 Zeitplan

Um das Spiel rechtzeitig zum Abgabetermin fertig zu bekommen und stets selbst ein Bild über den aktuellen Fortschritt zu haben, unterteilten wir das Projekt bereits relativ früh in drei Sprints und drei Meilensteine.

Für die Sprints legten wir dann die jeweiligen zu bearbeitenden Aufgaben fest, wobei sich diese natürlich über den Verlauf des Projekts teils änderten und andere Aufgaben hinzukamen. Beispielsweise kam die Idee einen Spielkommentator zu implementieren erst gegen Ende des 1. Sprints auf und wurde dann dem 2. Sprint als neue Aufgabe hinzugefügt.

In den drei Sprints arbeiteten wir jeweils auf den nächsten Meilenstein hin. Unser erster Meilenstein war es, bis Anfang Dezember 2021 einen ersten Spiel-Prototypen zu entwickeln der sich starten lässt und sich erste Konzepte ausprobieren lassen. Der erste Prototyp sollte dabei klar zeigen, in welche Richtung sich unser Spiel entwickelt. Bis zum zweiten Meilenstein Mitte Januar 2022, sollte die Entwicklung des Spiels abgeschlossen sein und höchstens nur noch kleine Verbesserungen und Bugfixes nötig sein. Der letzte Meilenstein ist die Projektabgabe am 9. Februar 2022. Zu diesem Datum sollten alle Verbesserungen abgeschlossen und keine Bugs mehr im Spiel vorhanden sein. Außerdem sollten auch alle anderen geforderten Projektdokumente abgabefertig sein.

Abbildung 1.1 zeigt den von uns festgelegten Zeitplan für das Projekt.

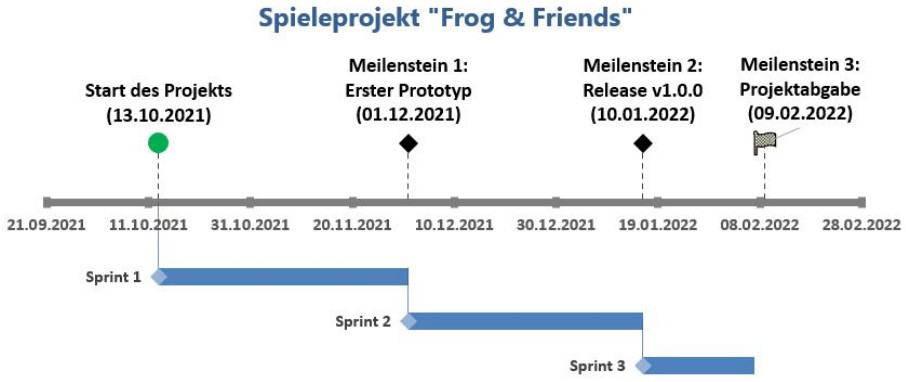


ABBILDUNG 1.1: Zeitachse des Projekts

Nachfolgend sind stichpunktartig die konkreten Aufgaben der einzelnen Sprints beschrieben. Einige der Aufgaben wurden von uns in Unteraufgaben (Tasks) unterteilt. Diese werden hier aus Platzgründen aber nicht separat angegeben.

Sprint 1

- Ideenfindung und erste Skizzen zum Leveldesign
- Erzeugung einer Tilemap und Tile-Palette
- Implementierung der Spieler-Animationen
- Implementierung von Spieler-Physik und Bewegung
- Implementierung des Spieler-Managers und dem lokalen Split-Screen-Modus
- Unterstützung von Gamepads als weiteres Eingabegerät
- Erstellung des ersten Test-Levels mit Terrain zum Testen der bereits implementierten Funktionen

Sprint 2

- Implementierung von Gegnern mit Logik und Animationen (Rhino, Mushroom, Plant)
- Implementierung von Hindernissen und Fallen mit Logik und Animationen (Rock Head, Saw, Falling Platform, Moving Platform)

- Hinzufügen einer grafischen Benutzeroberfläche (Hauptmenü, Pausenmenü, Anleitung, Matchresultat)
- Implementierung von Fruit-Items, Zählerskript und Darstellung auf Bildschirm
- Implementierung von Spawnpunkten und der Spawnlogik
- Implementierung des Spielkommentators
- Implementierung von Musik und Soundeffekten
- Implementierung der Todesanimation mit eigenen Spritesheets
- Fertigstellen des ersten Levels

Sprint 3

- Anfertigung der Projekt-Dokumentation
- Anfertigung einer Installations- und Spielanleitung
- Weitere kleine Verbesserungen und Bugfixes
- Erstellung von ausführbaren Builds für Windows, MacOS und WebGL

1.2 Details

1.2.1 Publikum und Ziel-Plattformen

Publikum

Auch wenn unser Spiel durch seine bunte und fröhliche Aufmachung und Musik zunächst den Eindruck eines kindergerechten 2D-Platformers vermitteln dürfte, ist es durch seine teils blutigen Animationen und den spöttischen Kommentator des Spielgeschehens kein Spiel für kleine Kinder. Die angestrebte Zielgruppe für das Spiel sind demnach vor allem Jugendliche und Erwachsene, die gerne Jump-n-Run-Spiele spielen und etwas Herausforderung suchen. Insbesondere ist das Spiel für diejenigen interessant, die sich gerne mit bis zu 3 weiteren Spielern lokal im Split-Screen miteinander messen möchten.

Ziel-Plattformen

Das Spiel steht als eigenständige Applikation für Windows und MacOS zur Verfügung. Zusätzlich lässt sich das Spiel unter dem Link `https://fabian12943.itch.io/frog-friends` als WebGL-Browserspiel in allen modernen Webbrowsern spielen. Dies erweitert die möglichen Plattformen auf denen sich das Spiel spielen lässt auf praktisch jedes internetfähige Endgerät, welches über einen WebGL-fähigen Browser verfügt und mit welchem sich eine Tastatur oder ein Controller verbinden lässt. Erfolgreich testen konnten wir die WebGL-Fähigkeit unter Windows und MacOS mit den Browsern Firefox und Chrome, sowie auf einem Android-Smartphone mit dem Chrome-Browser. Da das Spiel kein bestimmtes Seitenverhältnis erzwingt, lässt es sich bildschirmfüllend, ohne schwarze Ränder an den Seiten, auf jedem Bildschirm optimal anzeigen.

1.2.2 Features

Das Spiel verfügt über einige interessante Features, welche im Folgenden jeweils kurz in einer Formulierung aufgeführt werden, wie sie sich auch auf einer Spieleverpackung wiederfinden könnten.

Lokaler Splitscreen-Multiplayer für bis zu 4 Spieler

Das Spiel ermöglicht es bis zu vier Spielern gleichzeitig sich im Split-Screen Modus miteinander zu messen. Wer innerhalb der Spielrundenzeit mehr Items einsammelt und weniger oft stirbt gewinnt das Spiel.

Tödliche Gegner & tückische Fallen

Schießende Pflanzen, wütende Nashörner und giftige Pilze. Dazu noch rotierende Sägeblätter, herabfallende Plattformen und dich zertrümmernde Steine. Ohne das nötige Talent warten viele Tode und böse Kommentare.

Ein spöttischer Kommentator, dem du es nicht recht machen kannst

Ein Kommentator der dich und deine Spielweise nicht mag und es dich in jedem Satz wissen lässt, ganz egal ob du gerade gestorben bist, einen Punkt gemacht hast oder sogar gewonnen hast. Insgesamt verfügt das Spiel über 70 Sprüche, die mit Hilfe einer KI-Sprachsynthese-Webseite erstellt wurden.

Blutige Gewalt

Egal ob Spielercharakter oder Gegner: Alles was stirbt zerschmettert in viele, blutige Teile.

Support für Tastatur und Gamepads

Egal ob mit Tastatur, PS4-, PS5-, oder XBox-Gamepad, das Spiel lässt sich neben der Tastatur mit praktisch jedem aktuellen Gamepad spielen. Auch die Navigation in den Menüs ist mit dem Gamepad möglich.

Ein großes erstes Level

Das Spiel bietet bereits ein erstes Level, welches auch bei 4 Spielern noch groß genug ist und Herausforderungen an jeder Ecke bereit hält.

Anpassungsmöglichkeiten

Wenn der Kommentator wieder zu gemein war, lässt sich der Sound im Hauptmenü ausschalten oder leiser drehen. Ebenfalls kann die Rundenlänge zwischen 30 Sekunden und 5 Minuten nach Belieben eingestellt werden. Die Einstellungen bleiben auch nach dem Beenden des Spiels erhalten.

Kapitel 2

Umsetzung

In diesem Kapitel wird näher auf die praktische Umsetzung unserer Spielidee in der Laufzeit- und Entwicklungsumgebung Unity eingegangen. Vorab sei gesagt, dass etwaige verwendete Inhalte von Dritten in diesem Kapitel zwar namentlich genannt werden, die entsprechenden Links zu diesen allerdings erst in *Kapitel 4: Verwendete Assets* folgen. Dies soll der besseren Lesbarkeit und Übersichtlichkeit dienen.

2.1 Spieler-Charaktere

Da unser Spiel im lokalen Multiplayer von bis zu vier Spielern gleichzeitig gespielt werden kann, besitzt unser Spiel vier verschiedene Spieler-Charaktere. Diese werden bei Spielstart in zufälliger Reihenfolge an die Mitspieler zugewiesen.

Abbildung 2.1 zeigt alle vier zur Verfügung stehenden Spieler-Charaktere inklusive deren jeweiliger Name im Spiel.

ABBILDUNG 2.1: Spielbare Charaktere

In den nun folgenden Unterkapiteln 2.1.1 und 2.1.2 wird zunächst am Beispiel einer der vier Charaktere auf die Gestaltung und Animation sowie die Physik und Umsetzung der Bewegung eingegangen. Ebenfalls wird die *InputActions*-Komponente vorgestellt, die es leicht macht verschiedene Eingabegeräte, wie Tastaturen oder Controller für das Spiel zu nutzen. In Kapitel 2.1.3 wird anschließend noch auf das von uns erstellte *PlayerManager*-Prefab eingegangen, welches die Aufgabe hat, den Spielern das Beitreten zum Spiel mit ihrem jeweiligen Eingabegerät zu ermöglichen und ihnen dabei einen zufälligen, aber nicht doppelt vorkommenden Charakter zuzuweisen. Zusätzlich sorgt es dafür, dass nicht mehr Spieler beitreten können, als vor Spielstart ausgewählt wurden. Falls mindestens 2 Spieler beitreten, aktiviert es den Splitscreen, um den lokalen Multiplayer möglich zu machen.

2.1.1 Gestaltung und Animation

Die Grundlage für unsere Spieler-Charaktere und deren Animationen stellen die sogenannten Spritesheets des Unity-Assets *Pixel Adventure 1* dar. Unter einem Spritesheet versteht man ein Bild, das mehrere kleinere Bilder enthält, die in der Regel zusammengepackt werden, um

die Gesamtabmessungen zu verringern. Im Fall der Spielercharaktere benötigten wir bei allen Charakteren die Spritesheets für die Fälle *Idle (Verweilen)*, *Run*, *Jump* und *Fall*.

Abbildung 2.2 zeigt die von uns verwendeten Spritesheets am Beispiel des Charakters „Ninja Frog“.

ABBILDUNG 2.2: Spritesheets des Spieler-Charakters „Ninja Frog“

Diese Spritesheets trennten wir nun zunächst mithilfe des Unity-Sprite-Editors in einzelne Sprites (Einzelbilder). Mithilfe dieser konnten wir dann für alle vier Spieler-Charaktere die jeweils benötigten Animationen erstellen. Um das Abspielen dieser Animationen gezielt steuern zu können, erstellten wir den Animator-Controller (AC) *Player_AC*. Dieser enthält zwar selbst keine sichtbaren Animationen, stellt allerdings die Logik für alle spielbaren Charaktere bereit, wann welche Animation abgespielt werden soll. Die einzelnen Charaktere wiederum besitzen einen Animator-Override-Controller (AOC) und überschreiben darin die nicht sichtbaren Animationen des Player-Animator-Controllers mit ihren spezifischen animierten Sprites. Dieser Ansatz ermöglicht es, die Logik, welche für alle Spieler-Charaktere gilt, an einer einzigen Stelle zu modifizieren und für alle wirksam zu machen.

Abbildung 2.3 zeigt den Animator-Controller *Player_AC* (links), sowie die Verwendung eines Animator-Override-Controllers am Beispiel des Charakters „Ninja Frog“ (rechts).

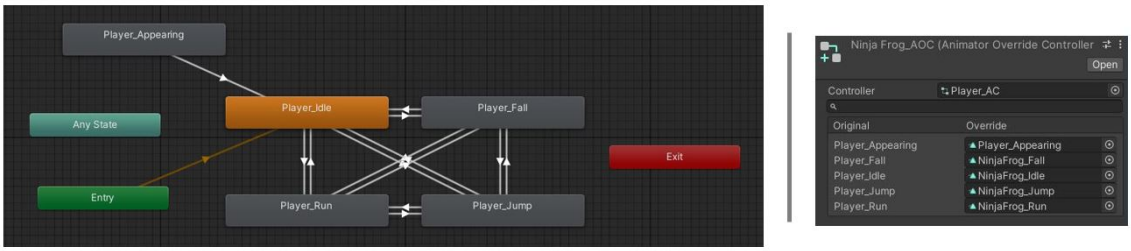


ABBILDUNG 2.3: Animator-Controller (links) und Animator-Override-Controller (rechts)

Um im fertigen Spiel auf verschiedene Ereignisse folgend die Animation zu wechseln, wurde der Animator-Komponente *Player_AC* der Parameter *state* hinzugefügt. Dieser kann vier verschiedene Integer-Werte annehmen und wechselt je nach Wert zwischen den Animationen *Idle*, *Run*, *Jump* und *Fall*. Diese Animation wird dann so lange in einer Schleife wiederholt, bis ein neuer Wert für *state* gesetzt wird. Wie bei 2D-Platformern üblich wird die Animation sofort gewechselt und besitzt keine Exit-Time und keinen Übergang (Transition) der Animationen

ineinander. Der Wert des Parameters *state* wird im Bewegungs-Skript der Spieler-Charaktere gesetzt und in *Kapitel 2.1.2: Physik, Bewegung und Input Actions* näher beschrieben.

Die Animation *Appearing*, welche beim Spawn oder Respawn eines Spielers abgespielt wird, wird nicht über den Parameter *state* gesteuert. Da diese Animation nicht in einer Schleife wiederholt werden soll, sondern nur jeweils ein einziges Mal, wird sie direkt aus dem Skript aufgerufen.

Da unser Spiel der Idee der etwas höheren Brutalität folgt und uns das Gegenstück zum in Abbildung 2.2 sichtbaren Spritesheet *Appear* beim Tod des Spielers nicht ausreichte, erstellten wir zusätzlich für alle Spieler-Charaktere ein Spritesheet, welches deren Körperteile im explodierten Zustand zeigt.

In Abbildung 2.4 sind die von uns erstellten Spritesheets hierzu abgebildet.

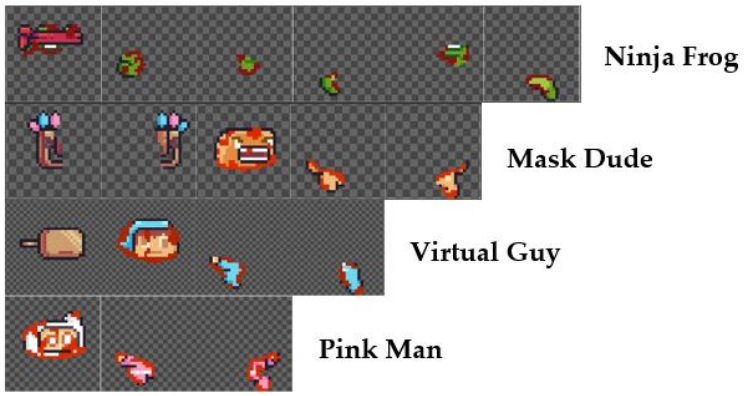


ABBILDUNG 2.4: Spritesheets der explodierten Spieler-Charaktere

Anders als bei den anderen Ereignissen sollten die einzelnen Sprites im Falle des Todes des Spielers natürlich nicht als Animation hintereinander angezeigt werden, sondern gleichzeitig wie bei einer Explosion in alle Richtungen davonfliegen.

Um dies zu erreichen, trennten wir die Spritesheets, wie oben bereits beschrieben, in ihre einzelnen Sprites und erzeugten dann aus jedem der Körperteile ein eigenes Prefab. Damit sich die einzelnen Körperteile später physikalisch korrekt verhalten, erhielten sie alle die Komponenten *Rigidbody 2D* und *Polygon Collider 2D*, sowie die Skripte *AddForce* und *SelfDestroy*. Das erstgenannte Skript sorgt dafür, dass das Körperteil direkt nach der Instanziierung im Spiel in eine zufällige Richtung weggeschleudert wird, das zweitgenannte Skript dafür, dass das Körperteil nach einer festgelegten Zeit zerstört wird und somit aus der Spielwelt verschwindet.

Alle Körperteile eines Spieler-Charakters zusammen dienen dann als Eingabeparameter für das ebenfalls von uns geschriebene Skript *PlayerExplode*. Dieses Skript wird ausgeführt, wenn der Spieler mit einem Objekt kollidiert, welches den Tag *Deadly* besitzt. Es deaktiviert die Sprite-Renderer-Komponente des Spielers, so dass dieser nicht mehr zu sehen ist, und instanziiert dann alle Körperteile in der Spielwelt an der Position des Spielers. Ebenfalls instanziiert wird das von uns erstellte Prefab *BloodSplash*, welches die Komponente *Particle System* verwendet, um physikalisch korrekte Blutspritzer zu erzeugen. Nach einer definierten Zeit wird der Spieler-Charakter an den nächsten Spawnpunkt gesetzt und die Sprite-Renderer-Komponente

wieder aktiviert.

2.1.2 Physik, Bewegung und Input Actions

Physik

Um es dem Spieler-Charakter zu ermöglichen, sich physikalisch korrekt durch die Spielwelt zu bewegen, fügten wir seinem Prefab die Komponente *Rigidbody 2D* hinzu. Mithilfe dessen konnte dem Charakter eine Masse sowie eine Schwerkraft hinzugefügt werden. Auch kann dort durch ein Einfrieren der Rotation auf der z-Achse verhindert werden, dass der Spieler umfällt, da dies in 2D-Spielen eher unüblich ist. Die Komponente *Box Collider 2D* sorgt dafür, dass der Spieler sowohl mit der Spielwelt, als auch mit anderen Objekten, wie Gegnern oder Hindernissen, kollidiert und nicht durch diese hindurch läuft oder fällt.

Bewegung

Die Möglichkeit für den Charakter sich in der Spielwelt nach links und nach rechts zu bewegen, sowie zu springen und die passenden Animationen dabei abzuspielen, wird ihm durch das Skript *Player Movement* gegeben, auf welches nun näher eingegangen wird.

Möchte der Spieler seinen Spielcharakter in der Welt bewegen und drückt auf die entsprechenden Knöpfe auf der Tastatur oder dem Controller, ruft die *Input Actions*-Komponente, welche etwas später in diesem Kapitel noch näher vorgestellt wird, in dem Skript die Funktion *Move(InputAction.CallbackContext context)* auf. Diese liest den übermittelten Wert für x aus dem Eingabegerät ein. Auf der Tastatur wäre dieser Wert beispielsweise *1*, wenn der Spieler auf die Taste zum rechts laufen drückt und *-1*, wenn der Spieler auf die Taste zum links laufen drückt. Bei einem Controller als Eingabegerät hingegen sind durch die Analogsticks auch Werte zwischen 0 und 1 bzw. -1 und 0 möglich und es kann die Geschwindigkeit des Charakters beeinflusst werden. Dieser Wert wird anschließend mit der definierbaren Variable *speed* multipliziert und als neue x-Koordinate des Spieler-Objekts gesetzt. Schon bewegt sich der Spieler. Ist dieser Wert ungleich 0, ist auch klar, dass der Spieler sich bewegt und der Parameter *state* des Animator-Controllers kann auf den Wert für die *Run*-Animation gesetzt werden. Andernfalls wird der Wert für die Animation *Idle* gesetzt. Auch kann durch diesen Wert die aktuelle Ausrichtung des Spielers geprüft werden und sich der Sprite des Spielers gegebenenfalls „flippen“ lassen.

Ähnlich wie bei der Bewegung des Charakters auf der x-Achse funktioniert das Springen. Drückt der Spieler den entsprechenden Knopf auf der Tastatur oder dem Controller, ruft die *Input Actions*-Komponente die Funktion *Jump(InputAction.CallbackContext context)* auf. Innerhalb der Funktion wird durch den boolischen Wert *context.performed* geprüft, ob der Knopf gedrückt wurde. Ist dies der Fall, wird die y-Koordinate des Spieler-Objekts auf die definierbare Variable *jumpingPower* gesetzt. Ebenfalls Verwendung findet der boolische Wert *context.canceled*. Dieser ist *True*, wenn man die Taste sehr schnell wieder losgelassen hat. In diesem Fall wird die y-Koordinate nur auf die Hälfte des Wertes der Variable *jumpingPower* gesetzt und das Spieler-Objekt macht nur einen halb so hohen Sprung. Damit der Spieler nicht unendlich viele Sprünge in der Luft aneinanderketten kann, ist vor jedem Sprung zu prüfen, ob der Charakter sich am Boden befindet. Hierfür verwenden wir die Methode *BoxCast()* der Unity-Klasse *Physics2D*. Diese gibt in unserem Fall den Wert *True* zurück, wenn die *BoxCollider2D*-Komponente

des Spieler-Objekts die Schicht (Layer) *groundLayer* berührt. Diese Schicht haben wir allen Objekten zugewiesen, von denen aus der Spieler abspringen können soll.

Listing 2.1 bildet die Implementierung der Jump-Funktion ab.

```
public void Jump(InputAction.CallbackContext context) {
  if (context.performed && IsGrounded()) {
    rb.velocity = new Vector2(rb.velocity.x, jumpingPower);
    GetComponent<PlayerAudioController>().PlayJumpSound();
  }
  if (context.canceled && rb.velocity.y > 0f) {
    rb.velocity = new Vector2(rb.velocity.x, rb.velocity.y * 0.5f);
  }
}
```

LISTING 2.1: Funktion zur Durchführung eines Sprungs des Spielers

Die letzte Funktion, die das Skript bereitstellt, ist es, jegliche Bewegungen der Charaktere einzufrieren, bis alle in der Welt gespawnt sind. Dies verhindert, dass manche Spieler schon anfangen können, bevor alle da sind.

Input Actions

Die Input-Actions-Komponente von Unity ermöglicht es schnell und einfach neue Eingabegeräte für das Spiel hinzuzufügen und hilft, die logische Bedeutung einer Eingabe von den physischen Mitteln der Eingabe (also der Aktivität auf einem Eingabegerät) zu trennen. Für unser Spiel haben wir beispielsweise eine sogenannte Action Map *Player* angelegt und dieser die Aktionen *Move* und *Jump* hinzugefügt. Diese beiden Aktionen sind alles, was der Spieler letztlich beim Spiel-Charakter steuern kann. Die Aktionen selbst kann man nun noch mit verschiedenen Eingaben verknüpfen (*binding*). Abbildung 2.5 zeigt die Input Actions für unser Spiel und lässt beispielsweise erkennen, dass der Spieler die Jump-Aktion auf der Tastatur mit der Leertaste und mit einem Controller wiederum mit einem Drücken auf den sogenannten *Button South* auslöst. Ebenfalls gibt es noch die Action Map *UI*, welche die Aktionen zur Steuerung der grafischen Benutzeroberfläche, wie dem Haupt- und Pausenmenü, über die Tastatur oder das Gamepad enthält.

Wie bereits in Listing 2.1 demonstriert wurde, können ausgeführte Input Aktionen in Skripten abgefangen und ausgewertet werden.

2.1.3 Player Manager

Das *PlayerManager*-Objekt ist eine sehr zentrale Komponente in unserem Spiel und erfüllt mehrere wichtige Aufgaben. Zum einen hat es die Aufgabe, den Spielern das Beitreten zum Spiel mit ihrem jeweiligen Eingabegerät zu ermöglichen und ihnen dabei einen zufälligen, aber nicht doppelt vorkommenden Charakter zuzuweisen. Zusätzlich sorgt es dafür, dass nicht mehr Spieler beitreten können, als vor Spielstart ausgewählt wurden. Falls mindestens 2 Spieler beitreten, aktiviert es den Splitscreen, um den lokalen Multiplayer zu ermöglichen. Im Folgenden wird näher auf die technische Umsetzung dieser Aufgaben eingegangen.

Um es überhaupt erst möglich zu machen, Spieler in einem Level spawnen zu lassen, benötigt das Objekt eine *Player Input Manager*-Komponente. In dieser Komponente lässt sich festlegen,

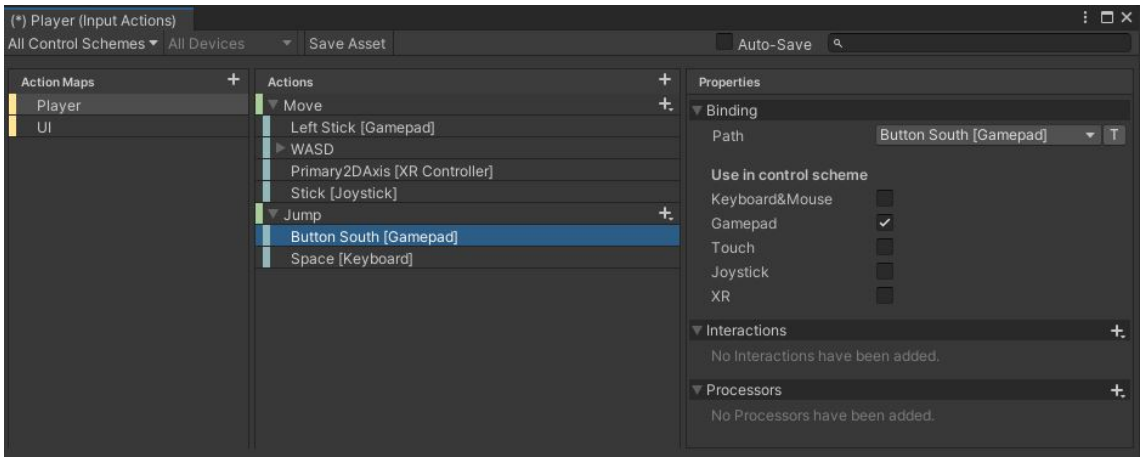


ABBILDUNG 2.5: Input-Actions zur Steuerung des Spielers

welche Eingaben auf den jeweiligen Eingabegeräten zu einer sogenannten *Join Action* führen und was in diesem Fall zu tun ist. Außerdem ist ein *Player Prefab* anzugeben, der dann initiiert (also gespawnt) wird. Auch kann in dieser Komponente festgelegt werden, ob der Splitscreen aktiviert wird, falls mehr als ein Spieler dem Spiel beitritt.

Um zu verhindern, dass jeder beitretende Spieler denselben, unter *Player Prefab* angegebenen Spieler zugewiesen bekommt, implementierten wir noch das Skript *CharacterSwitcher*. Dieses wird der *Player Input Manager*-Komponente als Event hinzugefügt und wird ausgeführt, sobald ein Spieler beitritt. Das Skript erhält von außen eine Liste mit allen vier möglichen Spieler-Prefabs zugewiesen und bringt diese noch vor Rendern des ersten Spieleframes in eine zufällige Reihenfolge. Dann weist sie der *Player Input Manager*-Komponente den ersten, nun zufälligen, Spieler-Prefab zu. Nach jedem Joint-Event weist sie der Input-Komponente das nächste Spieler-Prefab zu. Somit erhält jeder beitretende Spieler einen anderen, zufälligen Spieler-Charakter zugewiesen.

Um nun noch zu verhindern, dass mehr Spieler dem Spiel beitreten, als vor Spielbeginn angegeben wurde, deaktiviert das Skript *PlayerManagerController* die Möglichkeit zum Beitritt, sobald die angegebene Anzahl von Spielern beigetreten ist. Außerdem wird aus diesem Skript heraus das Match und der Timer gestartet.

Abbildung 2.6 zeigt das *PlayerManager*-Objekt mit all seinen Komponenten.

2.1.4 Kamera und Canvas

Um sicherzustellen, dass jeder Spieler seinen Charakter trotz des großen Levels stets gut im Blick hat, verfügt bei uns jedes Spieler-Prefab über eine eigene Kamera-Komponente. Damit das Level auch schon angezeigt wird, bevor ein Spieler spawnt, gibt es zusätzlich noch eine Hauptkamera. Diese wird allerdings deaktiviert, sobald der erste Spieler mit seiner eigenen Kamera spawnt. Zusätzlich wurde von uns das Skript *CameraFollow* geschrieben, welches dem Spieler bei seinen Bewegungen mit einem festlegbaren Offset folgt.

Da es auf dem Bildschirm sowohl Information darzustellen gibt, die jeden Spieler interessieren, sowie auch Informationen, die für jeden Spieler spezifisch sind und in erster Linie nur ihn

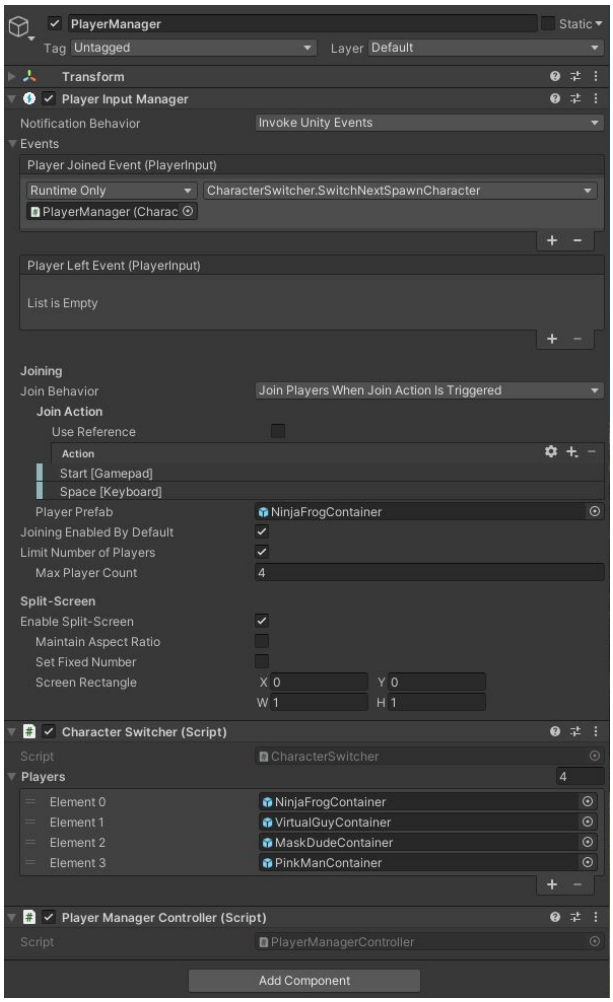


ABBILDUNG 2.6: Komponenten des *PlayerManager*-Objekts

betreffen, gibt es in unserem Spiel zwei verschiedene Canvas-Objekte.

Das erste Canvas trägt den Namen *General Canvas* und befindet sich auf der obersten Ebene. Die Informationen, die sich in diesem Canvas befinden, betreffen alle Spieler gleichermaßen und überlappen entsprechend alle Spieler-spezifischen Kameras. In diesem Canvas befindet sich neben dem sogenannten *Header*, bestehend aus verbleibender Zeit und einem Knopf zur Pausierung des Spiels, auch die kurze Spielerklärung, die zu Anfang jedes Levels angezeigt wird, das Pausenmenü und die Matchergebnisse.

Das zweite Canvas trägt den Namen *Player Canvas* und befindet sich als Komponente in jedem Spieler-Prefab. Die Informationen, die sich innerhalb diesen Canvas befinden, werden jedem Spieler innerhalb seines spezifischen Splitscreens angezeigt. In unserem Spiel wird in diesem Canvas angezeigt, wie viele Punkte der Spieler bereits erreicht hat.

Abbildung 2.7 zeigt die Bildschirmausgabe bei drei gespawnten Spielern und aktivem Pausenmenü

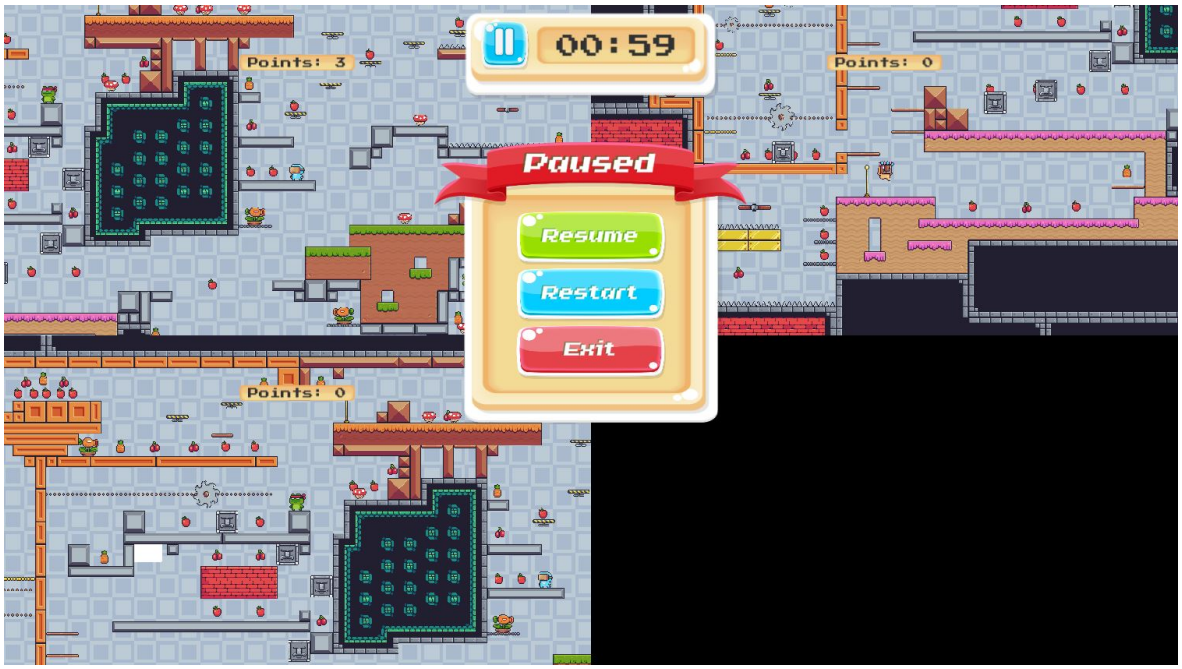


ABBILDUNG 2.7: Bildschirmausgabe bei drei Spielern und aktivem Pausenmenü

2.2 Nicht-Spieler-Elemente

2.2.1 Gegner

Insgesamt sind im Spiel-Prototyp zum aktuellen Zeitpunkt drei verschiedene Gegnertypen implementiert, welche für den Spieler tödlich sein können. Diese werden nachfolgend jeweils kurz vorgestellt. Alle Spritesheets für die Gegner wurden dem Unity-Asset *Pixel Adventure 2* entnommen.

Mushroom

Beim Gegnertyp *Mushroom* handelt es sich um kleine Pilze, welche sich in der Spielwelt auf und ab bewegen und für den Spieler tödlich sind, sollte er sie an den Seiten berühren. Dieser Gegnertyp lässt sich vom Spieler besiegen, indem dieser mittig von oben auf den Pilz springt. In diesem Fall zerspringt der Pilz in zwei Hälften und verschwindet vorerst aus der Spielwelt. Nach einer festgelegten Zeit respawnt der Pilz wieder an seiner ursprünglichen Position, um den Spielern auch im weiteren Spielverlauf eine Herausforderung zu bieten.

Der Sprite für den Pilz, sowie das Spritesheet für seine Laufanimation wurden dem Unity-Asset *Pixel Adventure 2* entnommen. Um den Pilz bei seinem Tod ähnlich explodieren zu lassen wie die Spieler-Charaktere, wurde von uns auch hier zusätzlich ein Spritesheet mit den blutigen Hälften des Pilzes erstellt. Die Erstellung der Animationen, sowie die Umsetzung der Explosion des Pilzes wurden sehr ähnlich wie bei den Spieler-Charakteren umgesetzt und werden daher hier nicht noch einmal näher beschrieben.

Für den Pilz, sowie für weitere in Zukunft zu erstellende patrouillierende Gegner, haben wir das Skript *AI Patrol* geschrieben. Dieses Skript sorgt dafür, dass die richtigen Animationen abgespielt werden und der Pilz bei nahenden Wänden oder Abgründen umdreht. Zur Erkennung von Wänden verwendet der Pilz seinen eigene *Box-Collider-2D*-Komponente und überprüft ständig auf eine Kollision mit Objekten, die der Schicht *groundLayer* angehören. Wird eine Kollision erkannt, so befindet sich eine Wand vor dem Pilz und dieser dreht um. Um zu erkennen, ob sich der Pilz kurz vor einem Abgrund befindet, verfügt das Pilz-Prefab zusätzlich über ein simples Transform-Objekt, welches sich etwas vor und unterhalb des eigentlichen Pilzes befindet. Auch bei diesem prüft das *AI Patrol*-Skript bei jedem Frame, ob dieses Objekt ein anderes Objekt der *groundLayer*-Schicht berührt. Ist dies nicht mehr der Fall, so steht ein Abgrund bevor und der Pilz dreht um.

Um unterscheiden zu können, wer bei einer Kollision von Spieler und Gegner stirbt und wer nicht, verfügt das Pilz-Prefab über zwei weitere *Box-Collider-2D*-Objekte. Der Collider mit dem Namen *SelfDeadDetect* deckt den Bereich ab, bei welchem der Pilz selbst stirbt. Wird dieser Collider vom Spieler berührt, wird das *Enemy Explode* Skript getriggert und der Pilz explodiert. Der Collider mit dem Namen *PlayerDeadDetect* und dem Tag *Deadly* deckt den Bereich des Pilzes ab, bei welchem bei einer Kollision der Spieler stirbt. In diesem Fall wird im Spieler-Prefab das *Player Explode* Skript getriggert und der Spieler explodiert.

Abbildung 2.8 zeigt bei beiden verschiedenen Box-Collider zur Unterscheidung der Folge eines Zusammenstoßes mit einem Spielercharakter.

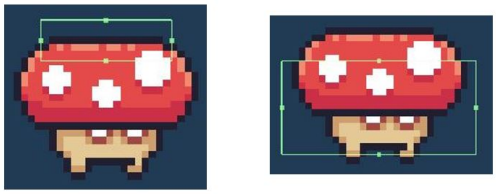


ABBILDUNG 2.8: Box-Collider *SelfDeadDetect* (links) und *PlayerDeadDetect* (rechts) des Mushroom-Gegnertyps

Rhino

Der Gegnertyp *Rhino* ist ein Nashorn, welches aufgebracht auf einer Ebene hin und her stürmt und bei Kontakt mit einer Wand eine kurze Crash-Animation abspielt, bevor es umdreht und weiter rennt. Dieser Gegnertyp lässt sich anders als der eben beschriebene *Mushroom* vom Spieler nicht besiegen, sondern ist bei jeder Berührung für den Spieler tödlich. Aus diesem Grund verfügt das Nashorn auch nur über den zusätzlichen Box-Collider *PlayerDeadDetect* mit dem Tag *Deadly*. Kollidiert das Spieler-Objekt mit diesem, so stirbt der Spieler.

Plant

Der *Plant* Gegner stellt eine Pflanze im Blumentopf dar, welche mit Bohnenkugeln aus ihrem Mund schießt. Dabei kann sich die Pflanze nicht von der Stelle bewegen und bildet somit ein stationäres Geschütz. Wird der Spieler von einer Bohnenkugel getroffen, ist dies tödlich für ihn. Gleichzeitig hat ein Spieler keine Chance einen Pflanzengegner zu besiegen.

Sowohl für die Pflanze als auch für die Bohnenkugel existieren zwei unterschiedliche Sprites, welche beide aus dem Unity-Asset *Pixel Adventure 2* stammen. Das Gleiche gilt auch für die Animationen der Pflanze.

Als Komponenten besitzt das *Prefab* der Pflanze zwei unterschiedliche *Box-Collider-2D*. Einer dient als generelle Abgrenzung des *GameObjects* zur Spielewelt. Der Andere hingegen fungiert als Prüfzone in welcher ermittelt wird, ob sich in dieser ein *GameObject* mit dem *Player*-Tag aufhält. Ist das der Fall wird die Animation der Pflanze von *Idle* auf *Attack* gesetzt und es werden Bohnenkugeln an der Position des Pflanzenmundes instanziiert. Tritt der Spieler aus der Prüfzone heraus, wird die Animation wieder auf *Idle* gesetzt. Dies geschieht durch das *Plant Logic* Skript, welches zusätzlich noch ermöglicht die Sekunden zwischen den Schüssen anzupassen. Somit ist die Feuerrate je nach Einsatzort einstellbar. Außerdem ist zu erwähnen, dass es sich bei der Bohnenkugel auch um ein eigenständiges *Prefab* handelt, welches im *Plant Logic* Skript referenziert wird.

ABBILDUNG 2.9: Pflanze während sie einen Schuss abgibt

Das Bohnenkugel *Prefab* besitzt das Tag *Deadly* und besitzt eine *Box-Collider-2D* Komponente. Somit ist diese tödlich für den Spieler, sollte er getroffen werden. Als weitere Komponente ist das eigens anfertigte *Bullet Logic* Skript hinzugefügt worden. Mit diesem lassen sich Fluggeschwindigkeit und Schussdistanz einer Kugel anpassen. Außerdem ist auch in dem Skript die *Destroy*-Funktion enthalten, welche das Kugel-*Gameobject* zerstört, sobald es seine definierte maximale Schussdistanz erreicht.

ABBILDUNG 2.10: Idle-Animation (oben) und Attack-Animation (unten)

2.2.2 Fallen und Hindernisse

Insgesamt sind im Spiel-Prototyp zum aktuellen Zeitpunkt vier verschiedene Fallen- und Hindernis-Typen implementiert. Diese werden nachfolgend jeweils kurz vorgestellt. Alle Spritesheets für die Fallen und Hindernisse wurden dem Unity-Asset *Pixel Adventure 1* entnommen.

Rock Head

Die *Rock-Head*-Falle ist den fallenden Steinblöcken aus dem bekannten *Super-Mario-Franchise* sehr ähnlich. Wie der Steinblock schwebt auch die *Rock-Head*-Falle in der Luft, kracht zu Boden und steigt dann wieder empor, wobei sich dieser Vorgang immer wieder wiederholt. Jedoch ist es der Rock-Head-Falle möglich sich in alle vier Richtungen (unten, oben, rechts und links) zu bewegen. Wird dabei der Spieler von der Falle getroffen oder von dieser eingequetscht, ist dies für ihn tödlich. Der Spieler hat keine Möglichkeit diese Falle zu zerstören.

Als Komponenten besitzt die *Rock-Head*-Falle einen *Box-Collider-2D*, sowie das *Rock Head Movement* Skript. Mit diesem wird die Beschleunigung, mit welcher sich die Falle bewegt, festgelegt. Dieser Wert wird im Skript genutzt, um die Geschwindigkeit der Falle zu berechnen. Dabei wird pro Frame die Beschleunigung auf die Geschwindigkeit aufaddiert, bis die Falle gegen ein Geländestück prallt. Anschließend wird die Geschwindigkeit auf null gesetzt. So entsteht eine natürliche Beschleunigung des Objekts.

Außerdem referenziert das Skript einen Array an *GameObjects*. In diesem werden die Wegpunkte (*engl. Waypoints*) für das Bewegungsmuster der Falle gespeichert. Mittels der *MoveTowards*-Funktion der *Vector2*-Klasse bewegt sich die *Rock-Head*-Falle zur Position des Wegpunktes. Ist deren Position nahezu identisch, wird der nächste Wegpunkt im Array als Ziel festgelegt. Gelangt man zum letzten Index des Arrays wird die Indexvariable auf null gesetzt, um somit zu garantieren, dass die Falle ihr Bewegungsmuster immer wieder abläuft.

Zusätzlich hat jedes *Rock-Head-Prefab* ein *PlayerDeadDetect GameObject*. Dieses definiert die Seite der Falle, welche bei Berührung tödlich für den Spieler ist. Dazu kommt eine sogenannte *Deadly Zone*. Diese markiert Bereiche, in welche der Spieler sterben kann, falls er von der Falle eingequetscht wird. Dadurch ist es dem Spieler möglich die Falle als Aufzug nach oben zu verwenden. Ist dieser aber währenddessen unaufmerksam, kann er leicht von der Falle zerquetscht werden.

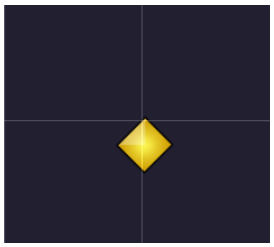


ABBILDUNG 2.11: Beispiel für einen Wegpunkt einer *Rock-Head* Falle

ABBILDUNG 2.12: *Rock-Head-Falle* mit dem *PlayerDeadDetect GameObject* und der *Deadly Zone*

Die *Rock-Head*-Falle besitzt fünf verschiedene Animationen. Eine *Idle*-Animation, in welcher sie blinkt und vier weitere welche abgespielt werden, sobald sie gegen das Gelände in der jeweiligen Richtung aufprallt.

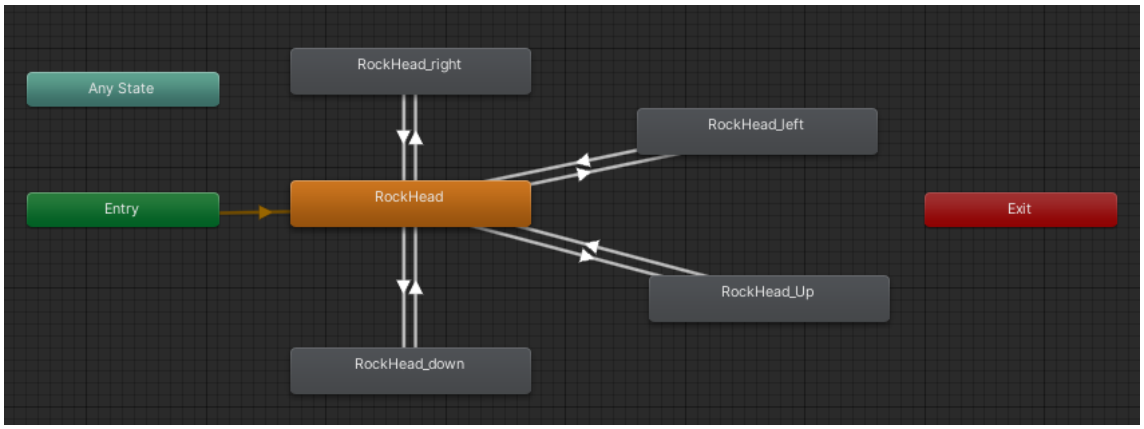


ABBILDUNG 2.13: Animator der *Rock-Head*-Falle

Saw

Es gibt zwei Einsatzformen der Sägen Falle. So gibt es eine stationäre Säge, welche an Wänden oder dem Boden angebracht werden kann und eine sich bewegende Säge. Diese kann einem vertikalen oder horizontalen Bewegungsmuster folgen, welche dem Spieler durch Verbindungsglieder einer Kette angezeigt wird. In beiden Fällen ist eine Berührung mit der Säge tödlich für den Spieler.

Da die stationäre Säge weniger Funktionalität benötigt, besitzt sie eine *Box-Collieder-2D* Komponente und ein *Rotate* Skript. Mit der Collider-Komponente wird geprüft, ob ein Spieler mit der Falle in Berührung kommt und mit dem Skript lässt sich flexibel die Geschwindigkeit, mit welcher die Säge rotiert, einstellen.

Neben den bereits im Prefab der stationären Säge enthaltenen Komponenten besitzt die sich bewegende Säge zusätzlich noch zwei weitere Skripte. Das *WaypointFollower*- und das*ChainCreator*-Skript. Bei ersterem handelt es sich um eine Abwandlung des *RockHeadMovement*-Skripts. Auch hier werden leere *GameObjects* als Wegpunkte referenziert, welche dann von der Säge in einer Schleife abgefahren werden. Mit dem*ChainCreator*-Skript werden die Verbindungsteile der Kette instanziiert. Dabei können Werte wie die Anzahl der Verbindungsteile und der Abstand zwischen den einzelnen Teilen festgelegt werden, sowie ob die Kette horizontal oder vertikal verlaufen soll.

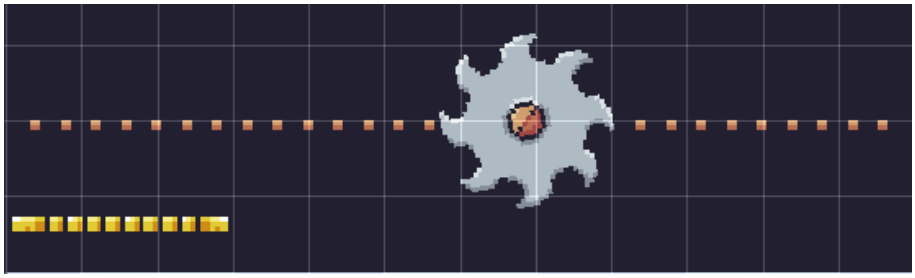


ABBILDUNG 2.14: Sich bewegende Sägen-Falle mit Kette

Fallende und sich bewegende Plattformen

Beide Plattformen stammen aus dem *Pixel Adventure 1* Assetpaket, besitzen aber verschiedene Eigenschaften. Die in Abbildung **??** links dargestellte Plattform stellt eine Falle für den Spieler dar. Diese fällt nach Berührung mit dem Spieler nach kurzer Zeit zu Boden und kann so für unaufmerksame Spieler tödlich sein. Die rechte Plattform hingegen bietet dem Spieler eine Transportmöglichkeit über vertikale und horizontale Distanzen.

Der *Prefab* der sich bewegenden Plattform besitzt die *Box-Collider-2D* Komponente, welche es dem Spieler ermöglicht sich auf die Plattform zu stellen. Damit dieser während der Fahrt der Plattform nicht hinunterfällt implementierten wir das *Sticky Platform* Skript. Durch dieses wird die Position des Spielers während einer Kollision mit der Position der Plattform synchronisiert. Entfernt sich der Spieler wieder von der Plattform, setzt sich seine Position zurück. Um das Bewegungsmuster der Plattform zu definieren, schrieben wir das *WaypointFollower*-Skript, welches eine Abwandlung des *RockHeadMovement*-Skripts ist, welches wir für die Bewegungsmuster der Steinblock Gegner verwenden. Auch hier werden leere *GameObjects* als Wegpunkte referenziert, welche dann von der Plattform in einer Schleife abgefahren werden.

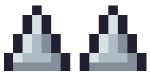
ABBILDUNG 2.15: Spieler auf einer sich bewegenden Plattform

Das *Prefab* der fallenden Plattform besitzt neben der *Box-Collider-2D* Komponente auch die *Rigidbody-2D*-Komponente, welche es ermöglicht einem *GameObject* physische Attribute zu geben. Diese Attribute werden mit dem eigens angefertigten *FallingPlatform* Skript manipuliert. Tritt der Spieler auf die Plattform wird nach einer definierbaren Zeit die *DropPlatform*-Funktion ausgeführt, welches das *Rigidbody* Attribut *isKinematic* des *GameObjects* auf *false* setzt und es somit zu Boden stürzen lässt und dann zerstört wird. Nach einer ebenfalls definierbaren Zeit erscheint die Plattform erneut.

```
private void OnCollisionEnter2D(Collision2D other){
    if (other.gameObject.CompareTag("Player")){
        Invoke("DropPlatfrom", timeToDrop);
        GetComponentInParent<ObjectRespawnerController>()
        .RespawnObject(spawnPosition, respawnDelay);
        Destroy(gameObject, timeToDelete);
    }
}
private void DropPlatfrom(){
    rb.isKinematic = false;
}
```

LISTING 2.2: Logik der fallenden Plattform

Spikes



Bei den Spikes handelt sich um ein Fallen-Objekt, welches den Spieler bei einer Kollision tötet und im fertigen Spiel an sehr vielen Stellen dafür sorgt, dass bestimmte Fehler des Spielers überhaupt erst eine tödliche Konsequenz haben. Beispielsweise lässt sich damit erreichen, dass das Herunterfallen von Plattformen oder das ungenaue Springen eines Spielers durch an Böden, Decken und Wänden angebrachten Spikes den Spieler tötet und ihm somit eine größere Herausforderung bietet. Auch werden die Spikes teilweise verwendet, um bestimmte Bereiche für den Spieler unzugänglich zu machen. Im Level werden die einzelnen Spikes zu Spike-Sets zusammengefasst und diese jeweils mit einer *Box-Collider-2D*-Komponente und dem Tag *Deadly* versehen. Bei einer Kollision des Spielers mit einem Objekt mit dem Tag *Deadly* wird dann das Skript *Player Explode* innerhalb des Spieler-Objekts ausgeführt und der Spieler explodiert.

2.2.3 Sammelbare Früchte

Die drei verschiedenen Früchte – ein Apfel, eine Kirsche und eine Ananas – sind die einzigen vom Spieler aufsammelbaren Gegenstände im Spiel. Dabei besitzt jede Frucht eine unterschiedlich hohe Punkteanzahl.

- Apfel = 1 Punkt
- Kirsche = 2 Punkte
- Ananas = 5 Punkte

Diese Punktzahl dient als Entscheidungswert dafür, welcher der Spieler das Spiel letztlich gewinnt. Diese Punktzahl wird während des gesamten Spiels über dem jeweiligen Spieler angezeigt. Die Sprites stammen aus dem Unity-Assetpaket *Pixel Adventure 1*.

Jedes der Früchte *Prefabs* besitzt einen *Box-Collider-2D*. Dies ermöglicht uns die *OnTriggerEnter2D*-Methode, um damit zu prüfen, ob das *GameObject* mit dem Spieler kollidiert. Ist das der Fall wird die *Collected*-Animation abgespielt.Nachdem diese abgespielt ist, wird das *GameObject* per *Destroy*-Methode zerstört.

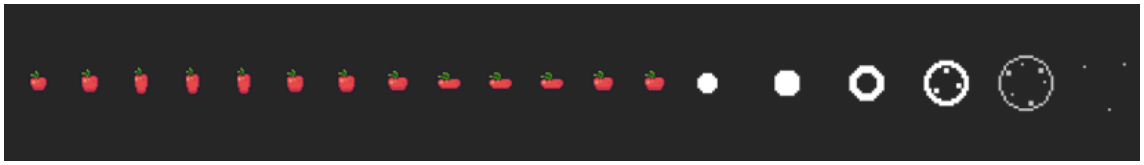


ABBILDUNG 2.16: Animation des Apfels beim Aufsammeln durch den Spieler

2.3 Levelgestaltung

Bei der Levelgestaltung verwendeten wir neben den von uns erzeugten Objekten aus *Kapitel 2.2 Nicht-Spieler-Elemente* zusätzlich noch diverse allgemeine Terrain-Spritesheets aus dem Unity-Asset *Pixel Adventure 1*, beispielsweise für Böden, Wände und den Hintergrund. Diese zerlegten wir wieder mithilfe des Unity-Sprite-Editors in ihre einzelnen Sprites und fügten diese einer sogenannten Tile-Palette hinzu. Aus dieser heraus lässt sich relativ bequem der statische Teil des Levels und der Hintergrund zusammenbauen.

Abbildung 2.17 zeigt die von uns verwendete Tile-Palette für die Terraingestaltung und den Hintergrund.

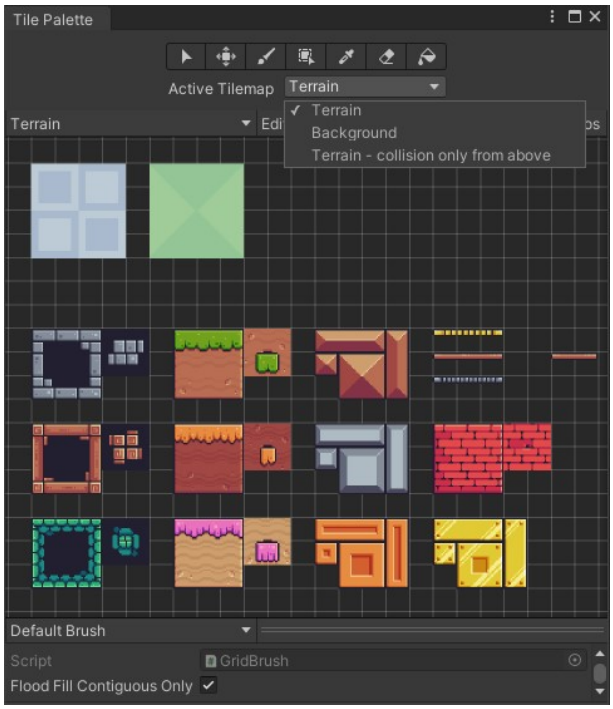


ABBILDUNG 2.17: Verwendete Tile-Palette für das Terrain und den Hintergrund

Aufgrund des benötigten Umfangs bei einem Level, welches von bis zu vier Spielern gleichzeitig bespielt werden soll, umfasst unser Spiel bisher nur ein Level. Dieses ist allerdings auch entsprechend groß. Im folgenden Kapitel soll dieses Level kurz vorgestellt werden.

2.3.1 Level 1

Im ersten Level unseres Spiels finden sich alle von uns implementierten Fallen, Hindernisse und Gegner wieder. Diese sind so im Level platziert und teilweise aneinandergereiht, dass sie dem Spieler eine möglichst große Herausforderung bieten. Die Items, die die Spieler sammeln, müssen um Punkte zu bekommen, sind so verteilt, dass nach besonders schwierigen Passagen wertvollere Items auf den Spieler warten, als in leichten Passagen. Bei der Erstellung des Levels wurde besonders darauf geachtet, dass die einzelnen Bereiche in keiner Sackgasse enden und der Spieler nicht den ganzen Weg zurück gehen muss. Außerdem wird so sichergestellt, dass die Spieler die Bereiche von beiden Seiten betreten können und sich keine Nachteile aus dem Spawnpunkt des Spielers ergeben. Die vier Spawnpunkte selbst sind so gesetzt, dass die Spieler direkt nach dem ersten Spawnpunkt möglichst weit weg voneinander sind und sich

nicht gegenseitig behindern.

Abbildung 2.18 zeigt die Ausmaße des ganzen Levels in der Szenenansicht von Unity.

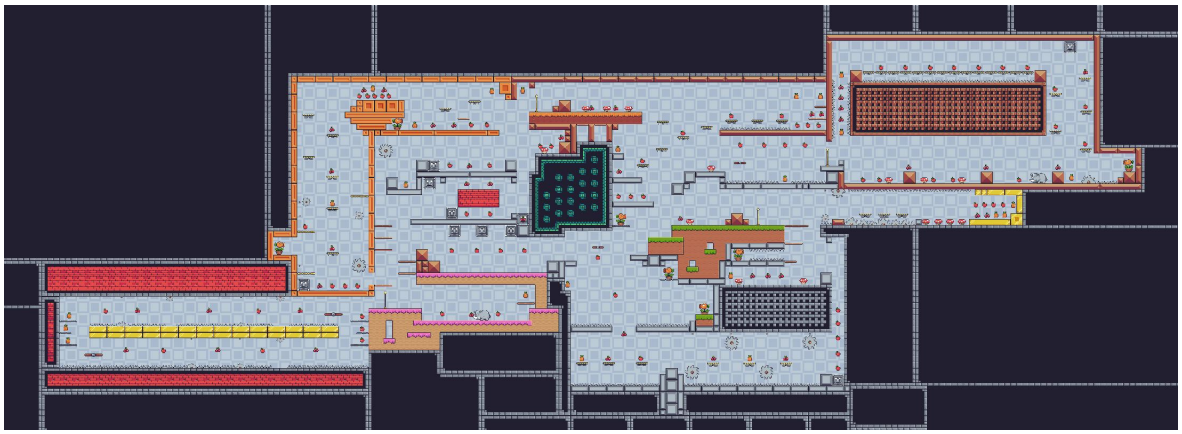


ABBILDUNG 2.18: Übersicht über das ganze 1. Level

2.4 Grafische Benutzeroberfläche

2.4.1 Hauptmenü

Für das Hauptmenü haben wir in Unity eine eigene Szene erstellt und ein eigenes kleines Minilevel erstellt, welches durch herumlaufende Gegner, sowie den animierten Helden des Spiels dafür sorgen soll, dass das Hauptmenü nicht so statisch und langweilig wirkt. Die Anzeige der Schaltflächen befinden sich in einem Canvas-Objekt und wird an einer Stelle angezeigt, an der sie das Mini-Level nicht überdeckt. Je nachdem auf welche Schaltfläche der Spieler drückt, werden nach Bedarf Schaltflächen ausgeblendet und andere Schaltflächen angezeigt.

Da uns keine der Standard-Schriftarten von Unity überzeugt hat, haben wir uns für die Aufschriften auf den Schaltflächen, sowie den Namen des Spieltitels eine lizenzfreie Schriftart aus dem Internet gesucht. Nach Umwandlung der Schriftart in ein „TextMeshPro“-Asset konnten wir die Schriftart zusätzlich in Unity weiter bearbeiten und so beispielsweise einen froschgrünen Schatten hinzufügen.

Um die getroffenen Einstellungen für die Lautstärke und die Rundenezeit unter „Settings“ auch nach einem Neustart des Spiels beibehalten zu können, haben wir die Unity-Klasse „PlayerPrefs“ verwendet, welche die Werte im Plattform-Register des Benutzers speichert.

Bedienen lässt sich das Hauptmenü und das im nächsten Kapitel beschriebene Pausenmenü sowohl mit der Maus, als auch mit der Tastatur oder dem Gamepad. Das aktuell ausgewählte Objekt ist hierbei immer etwas dunkler dargestellt, um die Bedienung zu erleichtern.

Abbildung 2.22 zeigt das Hauptmenü direkt nach Starten des Spiels.

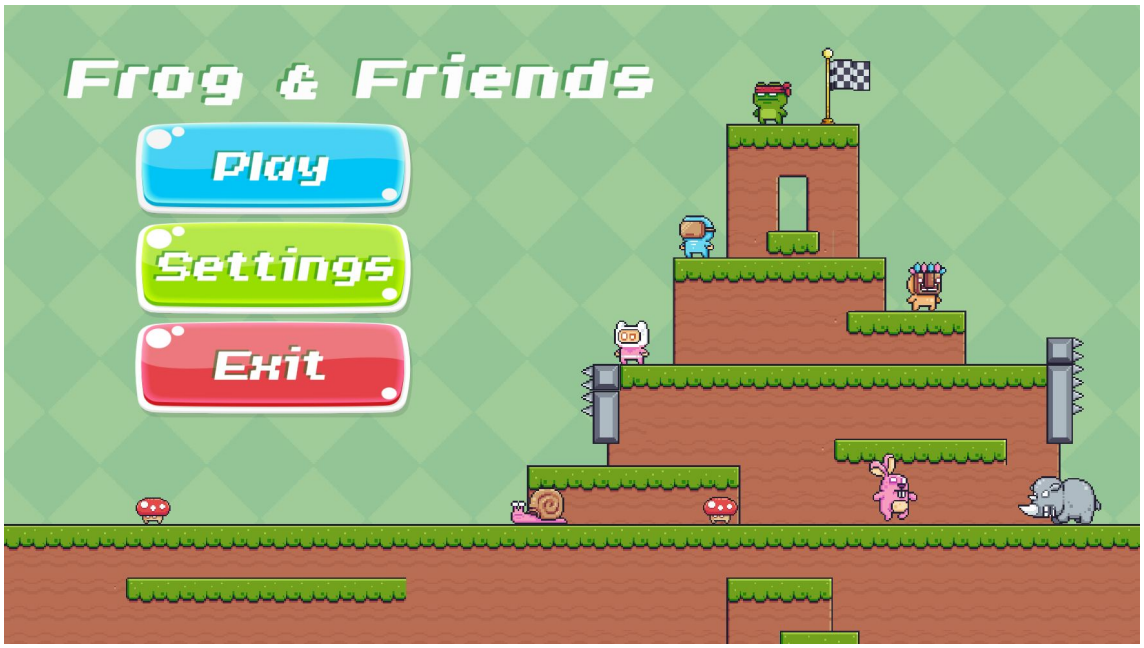


ABBILDUNG 2.19: Hauptmenü nach Starten des Spiels

2.4.2 Pausenmenü

Um das Spiel während der Spielrunde anzuhalten oder die Runde neu zu starten respektive zu beenden, lässt sich mit einem Klick auf das „Pause"-Symbol neben der verbleibenden Rundenzeit (oder der entsprechenden Taste auf Tastatur oder Gamepad) das Pausenmenü öffnen. Während das Pausenmenü geöffnet ist, ist die Variable timeScale der Unity-Klasse Time auf 0 gesetzt und das Spielgeschehen somit angehalten. Schließt der Spieler das Pausenmenü, wird die Variable wieder auf 1 gesetzt und das Spiel wird dort fortgesetzt, wo es gestoppt wurde. Vom Pausenmenü aus kann der Spieler die Runde auch neu starten oder beenden.

Abbildung 2.20 zeigt eine angehaltene Spielrunde mit geöffnetem Pausenmenü.

2.4.3 Match-Resultat

Sobald die Rundenzeit abgelaufen ist, werden durch die in Listing 2.3 gezeigte Methode die Statistiken der Spieler ausgewertet und die Spieler ihrer Leistung nach sortiert. Bei dieser Berechnung sind die erreichten Punkte des Spielers am wichtigsten. Bei selber Punktzahl wird derjenige besser platziert, der weniger Tode hatte. In dem unwahrscheinlichen Fall, dass Spieler dann immer noch gleich auf sind, wird alphabetisch sortiert. Die errechnete Platzierung wird dann an die Methode FillRow der Komponente Leaderboard übergeben, welche die Daten in die jeweiligen Zeilen einträgt. Ist dies geschehen, wird das Fenster, welches das Match-Resultat beinhaltet gerendert.

Abbildung 2.21 zeigt ein Match-Resultat nach einem Spiel mit zwei Spielern.

```
void CalculatePlayerScores() {
  List<GameObjec> players = new
     List<GameObject>(GameObject.FindGameObjectsWithTag("Player"));
```

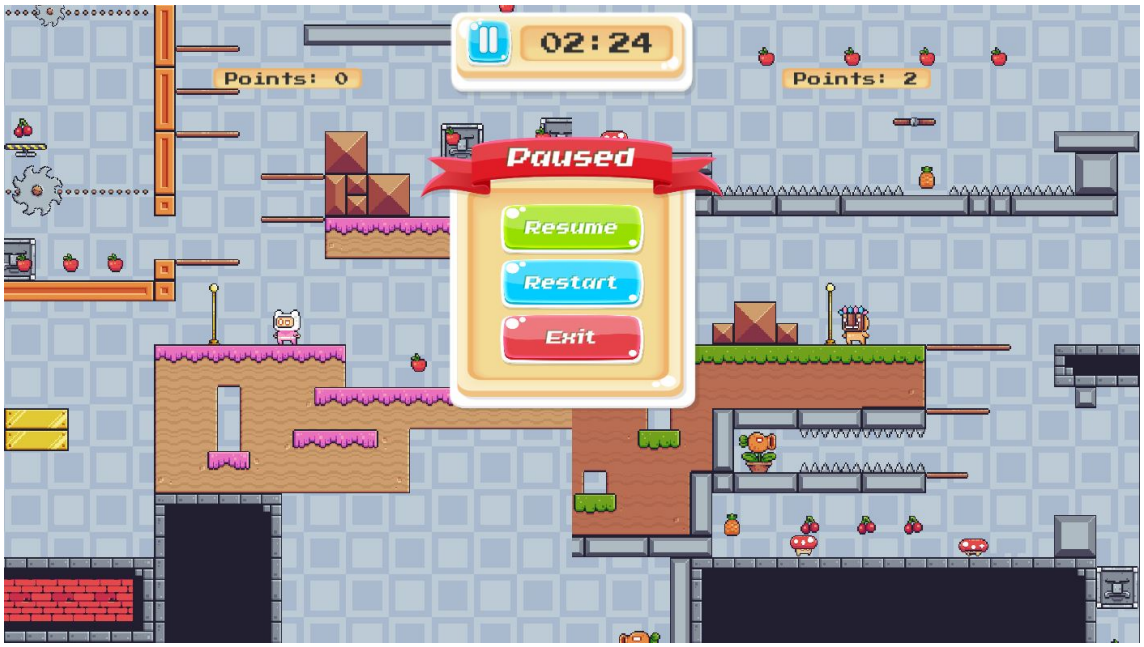


ABBILDUNG 2.20: Geöffnetes Pausenmenü während einer Spielrunde

```
  players = players.OrderByDescending(
            p=>p.GetComponent<PlayerStats>().points)
            .ThenBy(p=>p.GetComponent<PlayerStats>().deaths)
            .ThenBy(p=>p.name)
            .ToList<GameObject>();

  for (int i = 0; i < placesAmount; i++) {
    if (placesAmount < players.Count) {
      lbPlaces[i].GetComponent<Leaderboard>().FillRow(players[i]);
    } else {
      lbPlaces[i].GetComponent<Leaderboard>().FillRow(null);
    }
  }
}
```

LISTING 2.3: Funktion zur Berechnung der Spielerplatzierungen

2.5 Sound-Gestaltung

2.5.1 Soundeffekte

Im Spielgeschehen

Da unser Spiel im lokalen Multiplayer im Splitscreen von bis zu vier Spielern gleichzeitig gespielt werden kann, haben wir uns bewusst dazu entschlossen, die Anzahl und die Lautstärke der Soundeffekte möglichst zu begrenzen. Wir haben etwa die Geräusche beim Laufen der Spieler und Gegner sowie die Geräusche von verschiedenen Hindernissen nachträglich wieder entfernt, da diese bei einer höheren Spieleranzahl ständig zu hören waren und genervt haben. Die Lautstärke beim Springen der Spieler haben wir beibehalten, aber drastisch in der Lautstärke reduziert. Soundeffekte von Ereignissen, die nicht so oft auftreten, wie etwa das

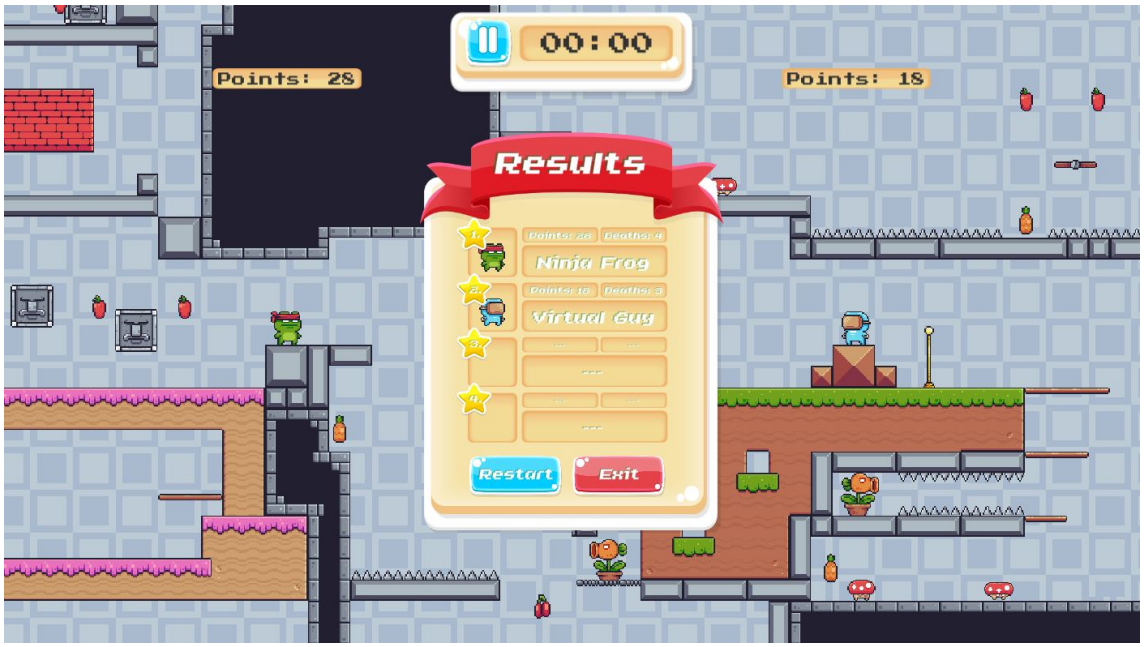


ABBILDUNG 2.21: Match-Resultat nach einem Spiel mit 2 Spielern

(Re-)Spawnen der Spieler, das Einsammeln eines Items oder die Explosion von Spielern oder Gegnern bei deren Tod, haben wir unverändert beibehalten. Um allerdings auch dort etwas Abwechslung zu garantieren, rufen manche Ereignisse einen zufälligen Soundeffekt aus einer bestimmten Gruppe auf. So wird bei der Explosion eines Spielers oder Gegners beispielsweise zufällig einer von drei verfügbaren blutigen Explosionsgeräuschen abgespielt. Beim Aufsammeln von Items entscheidet die Wertigkeit des Items über den abgespielten Sound. Je wertiger das Item ist, desto effektvoller ist der Soundeffekt.

In der Benutzeroberfläche

Auch die grafische Benutzeroberfläche im Haupt- und Pausenmenü verfügt über Soundeffekte. Diese beschränken sich allerdings auf das Abspielen eines Geräuschs beim Betätigen eines Buttons bzw. beim Loslassen eines Sliders.

2.5.2 Kommentator-Sprüche

Um das Spiel weiter von einem üblichen Jump-n-Run-Spiel abzuheben, haben wir einen Kommentator implementiert, welcher je nach aktuellem Geschehen verschiedene Kommentare über den Spieler und das Spielgeschehen abgibt. Dabei ist der Kommentator dem Spieler allerdings nicht sehr wohlgesonnen, sondern kritisiert den Spieler hauptsächlich auf zynisch-spöttische Art. Da die meisten Text-To-Speech-Systeme weder menschlich klingen, noch die passenden Stimmlagen für dieses Vorhaben anbieten oder die gewünschten Emotionen in der Stimme herüberkommen lassen, haben wir uns nach einer frei verfügbaren, KI-getriebenen Sprachsynthese-Technologie umgesehen und sind bei der Webanwendung unter `https://15.ai/` fündig geworden.

In der Webanwendung lässt sich aus diversen Charakteren aus der Film- und Spielwelt auswählen und nach Eingabe eines Textes verschiedene Sprachausgaben erzeugen. Durch die Angabe von Emotionen hinter dem auszusprechenden Text lässt sich auch ein direkter Einfluss

auf die Stimmlage und -betonung nehmen. Abbildung 2.22 zeigt die Verwendung des Tools am Beispiel des Zeichentrick-Charakters SpongeBob Schwammkopf.

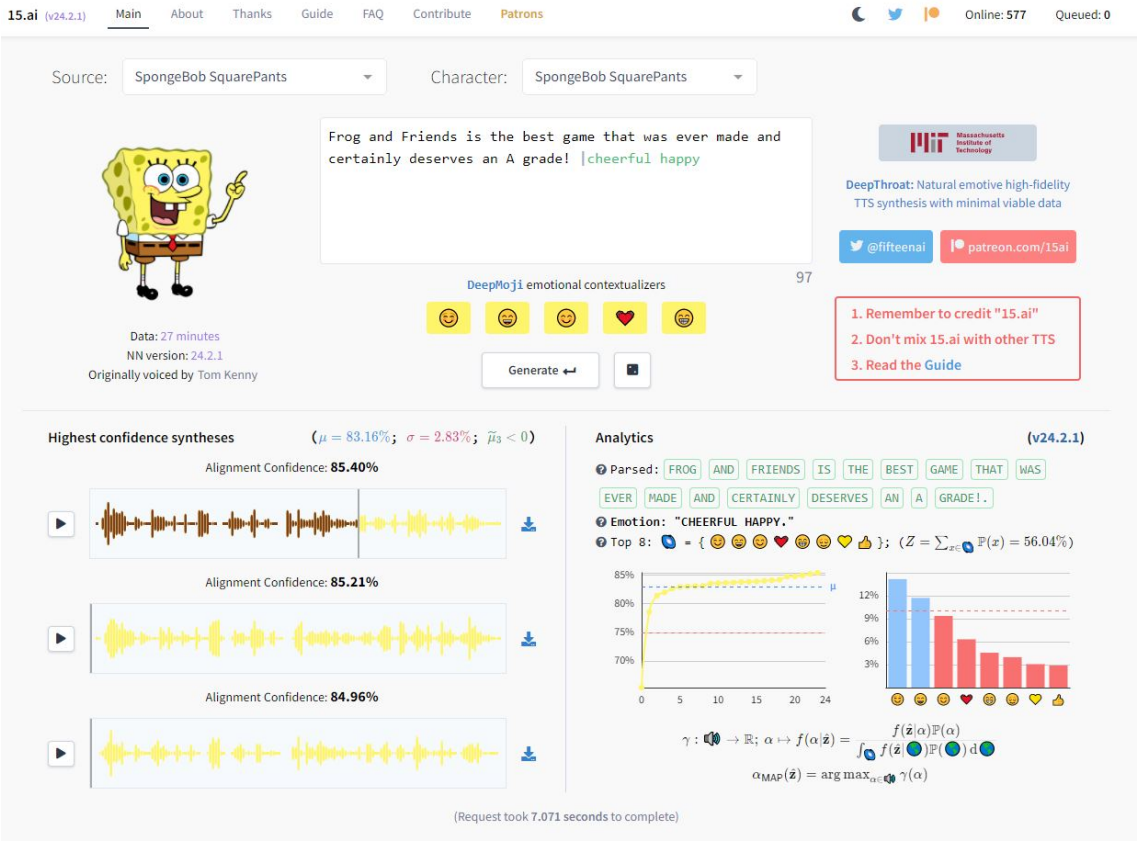


ABBILDUNG 2.22: Verwendung der Webanwendung zur Sprachsynthese unter https://15.ai/

Da wir für den Kommentator nach einer sehr lauten, tiefen Stimme gesucht haben, haben wir als Stimmquelle den Charakter „Heavy“ aus dem Videospiel „Team Fortress 2“ gewählt. Insgesamt haben wir uns für den Kommentator über 70 Sprüche ausgedacht, sie generiert, heruntergeladen und ins Spiel integriert. Dabei kommentiert der Kommentator das Spiel in folgenden Situationen:

- Das Level wird gestartet
- Verbleibende Spielzeit erreicht bestimmte Marke
- Der Spieler stirbt
- Der Spieler sammelt ein Item ein (verschiedene Sprüche für verschiedene Itemtypen)
- Der Spieler tötet einen Gegner
- Das Level ist zu Ende und die Ergebnisse und Spielerplatzierungen werden angezeigt

Neben den Kommentator-Sprüchen haben wir auch für die einzelnen spielbaren Charakter noch jeweils eine Audiodatei generiert, in welcher sie ihren eigenen Namen ausrufen. Dieser Ausruf findet pro Spieldurchgang allerdings nur einmal statt, wenn der jeweilige Charakter zum ersten Mal im Level gespawnt wird. Die hierfür verwendeten Stimmenprofile setzen sich folgendermaßen zusammen:

- „Ninja Frog“: Demoman (Team Fortress 2)
- „Virtual Guy“: Tenth Doctor (Doctor Who)
- „Mask Dude“: Dan (Dan Vs.)
- „Pink Man“: SpongeBob SquarePants (SpongeBob SquarePants)

2.5.3 Hintergrundmusik

Bei der Hintergrundmusik, welche im Hauptmenü und den einzelnen Levels läuft, haben wir uns bewusst für Musikstücke entschieden, die durch ihre fröhliche, heitere Art in einem kompletten Gegensatz zur für das Spielgenre unüblichen Brutalität des Spiels und dem spöttischen Spiel-Kommentator stehen. So soll das Spiel mit seiner grafischen und musikalischen Aufmachung zunächst an ein kindgerechtes, buntes Plattformer-Spiel erinnern und sich dem Spieler das wahre Gesicht des Spiels erst während des eigentlichen Spielens offenbaren.

Insgesamt verfügt das Spiel über vier verschiedene Musikstücke, welche durch ein Skript zufällig nacheinander abgespielt werden. Die Grundlautstärke der Hintergrundmusik ist deutlich leiser angesetzt, als die von Soundeffekten oder den Sprüchen des Kommentators, um diese nicht zu überschatten. Wie alle anderen Audioeffekte kann die Lautstärke der Hintergrundmusik im Hauptmenü verändert werden.

Kapitel 3

Ausblick

Für unseren Spiel-Prototyp war es uns wichtig alle von uns vorgesehenen Spielmechaniken zu implementieren und im ersten Level zu verwenden. Dies ist uns unserer Meinung nach auch gelungen. Dennoch gibt es natürlich einige Verbesserungsmöglichkeiten, Level-Erweiterungen und neue Spielmechaniken, welche in zukünftigen Versionen des Spiels umgesetzt werden können. Enige Ideen für die Zukunft sollen im Folgenden kurz beschrieben werden.

Weitere Levels

Da wir bereits für das erste Level viele Spielmechaniken (Skripte, Prefabs) implementiert haben, welche für jedes weitere Level ebenfalls benötigt werden, sollten sich weitere Level deutlich schneller erstellen lassen als das erste. Selbst ohne neue Gegnertypen, Hindernisse oder Spielmechaniken liesen sich so neue kreative Herausforderungen für die Spieler erstellen.

Weitere Gegnertypen und Hindernisse

Eine weitere Idee das Spiel in Zukunft um Inhalte zu erweitern, ist das Hinzufügen von neuen Gegnertypen und Hindernissen. Hierbei könnten die Gegnertypen beispielsweise andere Angriffsmuster oder Bewegungsmuster haben, als die bisherigen Gegner. Sowohl für weitere Gegner, als auch für weitere Hindernisse bieten die von uns verwendeten Unity-Assets *Pixel Adventure 1* und *Pixel Adventure 2* noch genügend Sprite-Sheets.

Neue Spielmechaniken

Auch über neue Spielmechaniken, die man in Zukunft hinzufügen könnte, haben wir uns Gedanken gemacht und sind dabei auf folgende Ideen gekommen:

- Spieler können sich gegenseitig beschießen und eliminieren und danach die Punkte des anderen klauen.
- Verschiebbare Kisten, mithilfe derer sich ansonsten zu hohe Ziele erreichen lassen und die Wege für andere Spieler blockieren lassen.
- Erweitern des Multiplayer-Modus, sodass Spieler auch über das Internet gegeneinander spielen können.

Zusätzliche Plattformen

Zwar lässt sich das Spiel bereits auf Windows und MacOS über eine native Applikation spielen, sowie durch den WebGL-Build auf vielen weiteren internetfähigen Endgeräten, doch gibt es auch hier noch Erweiterungspotential. So könnte man das Spiel in Zukunft auch noch nativ für Konsolen umsetzen oder das Spiel als App für Smartphones umsetzen. In letzterem Fall wäre es noch praktisch die Steuerung des Spiels so zu erweitern, dass nicht mehr zwingend eine Tastatur oder ein Gamepad mit dem Smartphone verbunden werden muss.

Kapitel 4

Verwendete Assets

4.1 Grafiken, Texturen und Schriftarten

Tileset „Pixel Adventure 1“:
https://assetstore.unity.com/packages/2d/characters/pixel-adventure-1-155360
Tileset „Pixel Adventure 2“:
https://assetstore.unity.com/packages/2d/characters/pixel-adventure-2-155418

Schriftart „Invasion 2000“: https://www.dafont.com/invasion2000.font
GUI Elemente Tilesets: https://opengameart.org/content/free-game-gui

4.2 Musik und Soundeffekte

4.2.1 Musikstücke

„Chunky Monkey“: https://bravewarrior.gumroad.com/l/YlULF
„Infinite Doors“: https://bravewarrior.gumroad.com/l/JZKWT
„Jumpy Game“: https://bravewarrior.gumroad.com/l/EYVTR
„Tiny Blocks“: https://bravewarrior.gumroad.com/l/KmsQPs

4.2.2 Soundeffekte

„Explosion_1“: https://www.youtube.com/watch?v=XXkmYgwux94
„Explosion_2“: https://www.epidemicsound.com/track/nIpTORYWGU/
„Explosion_3“: https://www.youtube.com/watch?v=0XKg_o1YfTg
„Jump“: https://freesound.org/people/cabled_mess/sounds/350906/
„Spawn“: https://freesound.org/people/RunnerPack/sounds/87043/
„Apple_Item“: https://freesound.org/people/bradwesson/sounds/135936/
„Cherry_Item“: https://freesound.org/people/Scrampunk/sounds/345297/
„Pineapple_Item“: https://freesound.org/people/Scrampunk/sounds/345299/
„Victory“: https://freesound.org/people/humanoide9000/sounds/466133/
„Button_Click“: https://freesound.org/people/Debsound/sounds/320549/

4.2.3 Text-To-Speech Sprüche

Alle Text-To-Speech Sprüche wurden mit der Webanwendung unter https://15.ai/ generiert.